CS305 – Programming Languages
Fall 2024-2025

HOMEWORK 1

Implementing a Lexical Analyzer (Scanner) for Logic Circuit Designer (LCD)

Due date: October 18th, 2024 @ 23:55

NOTE

Only SUCourse submission is allowed. No submission by e-mail. Please see the note at the end of this document for late submission policy.

1 Introduction

In this homework, you will implement a scanner for **LCD** language using flex. Your scanner will be used to detect the tokens in an LCD program. LCD is a scripting language (that is designed as an example language for CS305 homeworks) for logic circuit design automation.

LCD consists of three parts. The first part which is the declaration part starts with the declarations of inputs, the outputs, and the nodes. These declarations are done by using the keywords "**input**", "**output**", and "**node**" followed by the identifier names. They can be declared either alone or separated by commas. After the declarations, the assignment part follows. The nodes and the outputs are assigned to logical expressions of inputs, outputs and nodes. Then the last part starts; the evaluation part. The keyword "**evaluate**" is used in order to evaluate the designed configurations with different inputs. In the end, the program displays the value of the outputs.

An example LCD file is given below:

```
input X
input Y, Z
node A, B
node C
```

Node'lar yaplacak operasyonlar temsil ediyor
XOR AND gibi

```
output W, U
output T

A = X or Y
B = A xor Y
C = Z and Y

W = A and not Z
U = B and C
T = A or not C

evaluate circuit1 (X = true, Y = true, Z = false)
evaluate circuit2 (X = false, Z = true, Y = false)
```

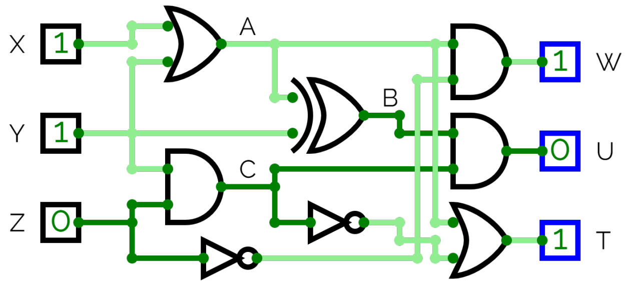

Figure 1: Sample Logical Circuit

Figure 1 shows the logical circuit designed by the example. It also gives the evaluation for the outputs under the values given by the first evaluate statement.

In an LCD program, there are keywords (e.g. evaluate), operators (e.g. and, xor), identifiers (programmer-defined names for variables, etc.), and punctuation symbols (e.g. ‘,’). Your scanner will catch these language constructs (introduced in Sections 2, 3, 4) and print out the token names together with their positions (explained in Section 6) in the input file. Please see Section 7 for an example of the required output. The following sections provide extensive information on the homework. Please read the entire document carefully before starting your work on the homework.

2 Keywords

Below is the list of keywords to be implemented, together with the corresponding token names.

Lexeme	**Token**	**Lexeme**	**Token**
`input`	tINPUT	output	tOUTPUT
node	tNODE	evaluate	tEVALUATE
identifier	tIDENTIFIER		

LCD is case–sensitive. Only the lexemes given above are used for the keywords.

3 Operators & Punctuation Symbols

Below is the list of operators and punctuation symbols to be implemented, together with the corresponding token names.

Lexeme	**Token**	**Lexeme**	**Token**
and	tAND	or	tOR
xor	tXOR	not	tNOT
true	tTRUE	false	tFALSE
(	tLPR	)	tRPR
=	tASSIGN	,	tCOMMA

4 Identifiers & Values

You need to implement identifiers and boolean values. Here are some rules you need to pay attention to.

- An identifier consists of any combination of letters, digits and underscore characters. However, it has to start with a letter.
- The token name for an identifier is tIDENTIFIER.
- Any character that is not a whitespace character and which cannot be detected as part of the lexeme of a token must be considered as an illegal character and must be printed out together with an error message (see below for example execution part about the error message details). The whitespace characters that should not be considered as illegal characters are, space or blank ( ), tab (`\t`), and newline (`\n`).

5 Comments

Comments in an LCD program are used to provide explanations, notes, or descriptions for the code and are not executed by the program. There are two ways to indicate comments:

- Single line comments: A comment that starts with // and extends up to the end of the line. Anything following // on the same line is considered a comment.

  Example:

```
// Initializing the inputs

input CS305, is, awesome

output I, love, programming, languages
```

Figure 2: An example LCD program (`test1.lcd`)

- Multiline comments: These comments begin with /* and end with */. Anything enclosed within this comment block is considered a multiline comment. This type of comment can span multiple lines. If you encounter another opening comment tag (/*) inside an existing multiline comment block, it should be considered as the start of a separate comment. The number of possible nested comments can vary; that is, it cannot be known in advance. Therefore, it is essential to treat each nested comment as a new comment block, each with its own content and corresponding closing tag (*/). A multiline comment does not have to have a closing tag. In such a case, your scanner does not complain about it and should treat the remainder of the program as if it's still within the comment block. A closing tag (*/) without a corresponding opening tag is not related to comments and should be considered as illegal characters.

  Example:

```
/*
    This is a multiline comment
    /*
        input A, B
        output E, F
        E = A and B
        F = A or B
        evaluate conf1 (A = true, B = false)
    */
*/
```

Figure 3: An example LCD program (`test2.lcd`)

A comment may appear on any line of a program. Anything that is commented out will be eaten up by your scanner silently. That is, no information will be produced for the comments or for the contents of the comments.

6 Positions

You must keep track of the position information for the tokens. For each token that will be reported, the line number at which the lexeme of the token appears is considered to be the position of the token. Please see Section 7 to see how the position information is reported together with the token names.

7 Input, Output, and Example Execution

Assume that your executable scanner (which is generated by passing your flex program through flex and by passing the generated lex.yy.c through the C compiler gcc) is named as LCDScanner. Then we will test your scanner on a number of input files using the following command line:

```
LCDScanner < test9.lcd
```

As a response, your scanner should print out a separate line for each token it catches in the input file. The output format for a token is given below for each token separately:

Detected	**Output**
identifier	⟨*row*⟩␣`tIDENTIFIER`␣`(`⟨*lexeme*⟩`)`
any other token	⟨*row*⟩␣⟨*token*⟩
illegal character	⟨*row*⟩␣`ILLEGAL_CHARACTER`␣`(`⟨*lexeme*⟩`)`

There are placeholders like ⟨*row*⟩ , ⟨*lexeme*⟩, ⟨*token_name*⟩ in the table above.

- ⟨*row*⟩ should be replaced by the location (line number) of the lexeme of the token.
- ⟨*lexeme*⟩ should be replaced by the actual lexeme of the token.
- ⟨*token_name*⟩ is the token name for the current item. These are the remaining, fixed–lexeme tokens, hence here you will use token names like `tNODE`, `tAND`, etc. based on what you detect.

Please note the use of underscore (_) characters as spaces in the output to be generated. **Please use no spaces, and use only a single underscore character as indicated in the output explanations and as given in the examples.** There will be no spaces inserted in the output by yourself. Instead, we will use underscore characters, and we will use only one underscore character to separate two pieces of information. Also, do not insert unnecessary newline characters. Each token must be on a separate line, and there should not be a line without any information. The reason for this is the following: We automatically compare your output with the correct output. Although it may seem harmless to insert a space, or an additional underscore in the output, an automatic comparison would indicate a difference between your output and the correct output, which can cost you some points in the grading.

As an example, let us assume that `test3.lcd` has the content as given in Figure 4:

```
input X, Y, Z
output W, U

W = (X or Y) not Z
U = (X and Z)

evaluate circuit1 (X = true, Y = false, Z = false)
```

Figure 4: An example LCD program (`test3.lcd`)

Then the output of your scanner must be:

```
1_tINPUT
1_tIDENTIFIER_(X)
1_tCOMMA
1_tIDENTIFIER_(Y)
1_tCOMMA
1_tIDENTIFIER_(Z)
2_tOUTPUT
2_tIDENTIFIER_(W)
2_tCOMMA
```

```
2_tIDENTIFIER_(U)
4_tIDENTIFIER_(W)
4_tASSIGN
4_tLPR
4_tIDENTIFIER_(X)
4_tOR
4_tIDENTIFIER_(Y)
4_tRPR
4_tNOT
4_tIDENTIFIER_(Z)
5_tIDENTIFIER_(U)
5_tASSIGN
5_tLPR
5_tIDENTIFIER_(X)
5_tAND
5_tIDENTIFIER_(Z)
5_tRPR
7_tEVALUATE
7_tIDENTIFIER_(circuit1)
7_tLPR
7_tIDENTIFIER_(X)
7_tASSIGN
7_tTRUE
7_tCOMMA
7_tIDENTIFIER_(Y)
7_tASSIGN
7_tFALSE
7_tCOMMA
7_tIDENTIFIER_(Z)
7_tASSIGN
7_tFALSE
7_tRPR
```

Note that, the content of the test files need not be a complete or correct LCD program. If the content of a test file is the following:

```
input AHMET, BEYZA, CEM
output DENIZ, EFE, FIGAN
"This is an illegal character: &"
// There is no circuit here.
```

Figure 5: An example LCD program (`test4.lcd`)

Then your scanner should not complain about anything and output the following information:

```
1_tINPUT
1_tIDENTIFIER_(AHMET)
1_tCOMMA
1_tIDENTIFIER_(BEYZA)
1_tCOMMA
1_tIDENTIFIER_(CEM)
2_tOUTPUT
2_tIDENTIFIER_(DENIZ)
2_tCOMMA
2_tIDENTIFIER_(EFE)
2_tCOMMA
2_tIDENTIFIER_(FIGAN)
3_ILLEGAL_CHARACTER_(")
3_tIDENTIFIER_(This)
3_tIDENTIFIER_(is)
3_tIDENTIFIER_(an)
3_tIDENTIFIER_(illegal)
3_tIDENTIFIER_(character)
3_ILLEGAL_CHARACTER_(:)
3_ILLEGAL_CHARACTER_(&)
3_ILLEGAL_CHARACTER_(")
```

Also note that some of the test cases may have nested comment lines in that case you should handle them. If the content of a test file is the following:

```
/*
    This is a multiline comment starting

input A, B, C
output D, E, F

D = (A and B) xor C
E = ((R or B) not C) and (A xor B)
F = (S xor B) or (A and C)

    This is a multiline comment ending
*/

evaluate X(A = false, B = false, C = false)
```

Figure 6: An example LCD program (test5.lcd)

Then your scanner output should be the following:

```
14_tEVALUATE
14_tIDENTIFIER_(X)
14_tLPR
14_tIDENTIFIER_(A)
14_tASSIGN
14_tFALSE
14_tCOMMA
14_tIDENTIFIER_(B)
14_tASSIGN
14_tFALSE
14_tCOMMA
14_tIDENTIFIER_(C)
14_tASSIGN
14_tFALSE
14_tRPR
```

8 How to Submit

Submit only your flex file (**without zipping it**) on SUCourse. The name of your flex file must be:

username-hw1.flx

where username is your SUNet username. For example, if your SUNet user name is `husnu.yenigun` then your file must be named as `husnu.yenigun-hw1.flx`.

9 Notes

- **Important**: Name your file as you are told and **don't zip it.**
  [-10 points otherwise]
- Do not copy-paste LCD program fragments from this document as your test cases. Copy/paste from PDF can create some unrecognizable characters. Instead, all LCD code fragments that appear in this document are provided as a text file for you to use.
- Make sure you print the token names exactly as it is supposed to be. Also, make sure you use underscore characters (not spaces) as explained above. You will lose points otherwise.
- Please keep you have the access to the golden in your mind. The golden will print your test cases as it is wanted. Therefore, you can compare your scanners' output with the goldens output.
- In order to use the golden for the given test cases, you can call this command:
  ```
  /home/osmankara/cs305/goldens/hw1-golden <
  /home/osmankara/cs305/tests/hw1-tests/testX.lcd
  ```
  Please keep in mind the `X` is the number of the test case.
- No homework will be accepted if it is not submitted using SUCourse.
- You may get help from our TA or from your friends. However, **you must write your flex file by yourself**.
- Start working on the homework immediately.

- If you develop your code or create your test files on your own computer (not on cs305.sabanciuniv.edu), there can be incompatibilities once you transfer them to cs305.sabanciuniv.edu. Since the grading will be done automatically on cs305.sabanciuniv.edu, we strongly encourage you to do your development on cs305.sabanciuniv.edu, or at least test your code on cs305.sabanciuniv.edu before submitting it. If you prefer not to test your implementation on cs305.sabanciuniv.edu, this means you accept to take the risks of incompatibility. Even if you may have spent hours on the homework, you can easily get 0 due to such incompatibilities.

LATE SUBMISSION POLICY

Late submission is allowed subject to the following conditions:

- Your homework grade will be decided by multiplying what you get from the test cases by a "submission time factor (STF)".
- If you submit on time (i.e. before the deadline), your STF is 1. So, you don't lose anything.
- If you submit late, you will lose 0.01 of your STF for every 5 minutes of delay.
- We will not accept any homework later than 500 minutes after the deadline.
- SUCourse+'s timestamp will be used for STF computation.
- If you submit multiple times, the last submission time will be used.